CSC384 Notes

Jenci Wei

Fall 2022

Contents

1 Introduction

1.1 Definitions of AI

Four Definitions

Cognitive Modeling Systems that think like humans	**Laws of Thought** Systems that think rationally
Turing Test Systems that act like humans	**Rational Agent** Systems that act rationally

Cognitive Modelling (Thinking Humanly)

- Want to think like humans because humans is one of a few examples of intelligence
- How humans think
  - Introspection
  - Psychological experiments
  - Brain imaging (e.g. MRI)
- Cognitive science: develop precise and testable theories of the human mind using theoretical models and experiments

Turing Test (Acting Humanly)

- Operational definition: if the interrogator cannot distinguish the entity from a human, then the entity passes the test and is considered intelligent
- Usefulness of the test is questionable
  - Allows us to recognize intelligence, but does not provide a way to realize intelligence
  - Gave rise to core areas of AI: NLP, KR, ML, CV, robotics, etc.

Rationality

- **Rationality**: an abstract "ideal" for intelligence, rather than "whatever humans do"
- A system is rational if it does the "right thing" given what it knows

Laws of Thought (Thinking Rationally)

- Formalized correct thinking
- The logicist tradition
  - Goal is to express our knowledge using logic
  - A system with such knowledge can solve any problem
- Two obstacles
  1. Difficult to translate natural language to logic
  2. Brute force search through the large number of statements is too slow

Rational Agent (Acting Rationally)

- A rational agent acts to achieve the best (expected) outcome
- Rational behaviours:
  - Create and pursue goals
  - Operate autonomously
  - Perceive environment
  - Learn
  - Adapt to changes

Choice of Definition: Rational Agent

- We care about behaviour instead of thinking and reasoning
  - Acting rationally is more general than thinking rationally (i.e. goal for thinking rationally is to act rationally)
  - No logically correct thing to do, yet we still need to take an action
  - Want to act quickly without thinking
- We measure success against rationality instead of against humans
  - Humans often act in ways that we don't consider intelligent (i.e. "predictably irrational")
  - Rationality is mathematically well-defined and completely general

2 Uninformed Search

2.1 Generic Search Algorithm

Pseudocode:

```
Search(Initial State, Successor Function, Goal Test)
   Frontier = { Initial State }
   While Frontier is not empty do
      Select and remove state Curr from Frontier
      If Curr is a goal state
         return Curr
      Add Successors of Curr to Frontier
   Return no solution
```

Recovering the path from the initial state

1. Let the frontier store the *paths* instead of the *states*
2. Let each node store a *reference to its parent node*

Search Graph

- Well-defined given an initial state and a successor function
- Static object
- Contains all the states
- Every state appears once
- Arrows may go in both directions
- A finite graph may produce an infinite tree

Search Tree

- Constructed during search execution
- Grows dynamically as search progresses
- A state may appear multiple times
- Some state never appears
- Arrows go in one direction only

Uninformed Search Strategies

- The order in which we remove nodes from the frontier determines our search strategy and its properties
- Need a *fixed rule* for selecting the next state to be expanded
- Popular uninformed search techniques:
  - Breadth First Search
  - Uniform Cost Search
  - Depth First Search
  - Iterative Deepening Search

Properties of Search Algorithms

- *Completeness*: will the search always find a solution if a solution exists?
- *Optimality*: will the search always find the least-cost solution?
- *Time Complexity*: what is the maximum number of nodes that we must expand?
- *Space Complexity*: what is the maximum number of nodes that we must store in memory?

Quantities

- b: branching factor
  - Maximum number of successors of any state
- d: depth of the shallowest goal node
  - Also the length of the shortest solution
- m: max depth of the serach tree
  - Also the length of the longest path

2.2 Breadth First Search

The frontier is a *queue* (FIFO)

- Expands the oldest node added to the frontier

Completeness

- Yes
- Must find the shallowest goal node at depth d if b and d are finite

Optimality

- No
- Guaranteed to find the shallowest goal node, which is not necessarily the least-cost solution
- If step costs are the same, then BFS is optimal
- BFS does not consider path costs

Time Complexity

- $\mathcal{O}(b^d)$
- Must visit the top d levels
  - Each level has b^0 (root, level 0), b^1 (level 1), ..., b^d (level d) nodes, respectively
  - Total number of nodes explored: $1 + b + b^2 + \cdots + b^d \in \mathcal{O}(b^d)$

Space Complexity

- $\mathcal{O}(b^d)$
- Visits the top d levels in the worst case
- Whenever we go down a level, the size of frontier grows by a factor of b
- The frontier is the largest when we reach level d, the number of nodes $\leq b^d$

2.3 Uniform Cost Search

The frontier is a *priority queue* ordered by path cost

- Expands the least-cost path in the frontier
- Also known as *Dijkstra's shortest path algorithm*
- Identical to BFS if each action has the same cost

Completeness

- Yes, under mild conditions
- The branching facrtor b is finite
- The cost of every action is bounded below by a positive constant $\varepsilon > 0$

Optimality

- Yes under mild conditions
- The branching factor b is finite
- The cost of every action is bounded below by a positive constant $\varepsilon > 0$

Time and Space Complexity

- $\mathcal{O}(b^{C^*/\varepsilon})$
  - Greater than $\mathcal{O}(b^d)$ since the optimal path may not be the shortest path
- C^* is the cost of the optimal solution
- ε is the minimal cost of an action
- How did we get that expression?
  1. Total cost is C^*
  2. Cost of each step $\geq \varepsilon$
  3. Total cost $\div$ cost of each step = number of steps
  4. Total cost $\div$ min cost of each step $\geq$ number of steps
  5. Therefore number of steps $\leq C^*/\varepsilon$
  6. Since number of steps corresponds to the depth of the search tree, we have $\mathcal{O}(b^{C^*/\varepsilon})$

2.4 Depth First Search

The frontier is a *stack* (LIFO)

- Expands the last/most recent node added to the frontier

Completeness

- No
- May get stuck in an infinite path
- May get stuck in a cycle

Optimality

- No
- DFS does not consider path costs

Space Complexity

- $\mathcal{O}(b^m)$
- Explores a single path at a time
- Remembers m nodes on the current path, and at most b siblings for each node

Time Complexity

- $\mathcal{O}(b^m)$
- Visits the entire search tree in the worst case

2.5 Iterative Deepening Search

Depth Limited Search

- Performs DFS up to a pre-specified depth limit L
- Prevents DFS from getting stuck in an infinite path
- Complete iff a solution of length $\leq L$ exists

Iterative Deepening Search

- For every depth limit L (starting from 0), perform DLS up to depth L, until we find a solution or the DLS explores the entire search tree
- Every time the depth limit increases, we start the DLS from scratch
  - Cannot reuse work done for previous depth limit, else we would have a space complexity comparable to BFS

Completeness

- Yes
- IDS is complete in a way similar to BFS, where it explores the tree level by level until it finds a goal

Optimality

- No
- Similar to BFS, IDS does not consider path costs and is guaranteed to find the shallowest goal node

Space Complexity

- $\mathcal{O}(bd)$
- Terminates at level d
- Keeps track of one path of length at most d
- Remembers at most b siblings of the node at each level

Time Complexity

- $\mathcal{O}(b^d)$
- Same as BFS
  - BFS: $b^0 + b^1 + \cdots + b^d \in \mathcal{O}(b^d)$
  - IDS: $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d \in \mathcal{O}(b^d)$
    - Which is $(b^0) + (b^0 + b^1) + (b^0 + b^1 + b^2) + \cdots + (b^0 + b^1 + \cdots + b^d)$

2.6 Multiple Path Pruning

It is useless to remember multiple paths to the same node

- If we have already found a path to a node, we can discard other paths to the same node

Generic Search Algorithm with Multiple Path Pruning

```
Search(Initial State, Successor Function, Goal Test)
   Frontier = { Initial State }
   Explored = { }
   While Frontier is not empty do
      Select and remove state Curr from Frontier
      If Curr is NOT in Explored
         Add Curr to Explored
         If Curr is a goal state
            return Curr
         Add Successors of Curr to Frontier
   Return no solution
```

Effects on Space Complexity

- Incurs high space complexity
- Combining wth BFS:
  - Stores the entire tree up to depth d in the worst case, which is $\mathcal{O}(b^d)$
  - Same asymptotic space complexity as BFS, which is $\mathcal{O}(b^d)$
- Combining with DFS:
  - Stores the entire tree in the worst case, which is $\mathcal{O}(b^m)$
  - Way worse than plain DFS, which is $\mathcal{O}(bm)$

Effects on Optimality

- UCS with multi path pruning is *still optimal*
  - The first path to each state found by UCS must be the lowest-cost path to that state
  - The pruned paths *cannot* have lower costs than the first path UCS found
- Optimality may no longer hold for some heuristic search algorithms... (next chapter)

3 Heuristic Search

3.1 Motivation

Uninformed Search

- Has no problem-specific knowledge
- Treats all the states in the same way
- Does not know which state is closer to a goal

Heuristic Search

- Has problem-specific knowledge, i.e. a *heuristic*
- Considers some states to be more promising than others
- Estimates which state is closer to a goal using the heuristic

Heuristic Function

- A search heuristic $h(n)$ is an *estimate* of the cost of the *cheapest* path from node n to a goal node
- A good $h(n)$ has the properties below:
  - Problem-specific
  - Non-negative
  - $h(n) = 0$ if n is a goal node
  - Can compute $h(n)$ easily without search

Functions

- Suppose we are at state n
- The *cost function* $g(n)$ is the cost of the path from the initial state to the current state n
- The *heuristic function* $h(n)$ is the estimate of the cheapest path from the current state n to a goal state
- Search algorithms:
  - UCS expands the state with the lowest $g(n)$
  - GBFS expands the state with the lowest $h(n)$
  - A* expands the state with the lowest $f(n) = g(n) + h(n)$

3.2 Greedy Best First Search

Frontier is a priority queue ordered by $h(n)$

- Expands the node with the smallest $h(n)$

Completeness

- No
- Poor choice of heuristic

Optimality

- No
- Poor choice of heuristic

Space and Time Complexity

- Both are exponential
- If $h(n) = 0$ for every state n, then GBFS is equivalent to an uninformed search algorithm

3.3 A* Search

Frontier is a priority queue ordered by $f(n) = g(n) + h(n)$

- Expands the node with the smallest $f(n)$
- Cost that is already spent + estimated cost that needs to be spent

Space and Time Complexity

- Both are exponential

Completeness and Optimality

- Yes to both, if $h(n)$ is *admissible*

Admissible Heuristic Definition

- A heuristic function $h(n)$ is *admissible* iff it *never overestimates* the cost of the cheapest path from state n to a goal state
- Let $h^*(n)$ denote the cost of the cheapest path from state n to a goal state. Then an admissible $h(n)$ satisfies

$$\forall n, 0 \leq h(n) \leq h^*(n)$$

Admissible Heuristic Intuition

- It is a lower bound on the actual cost
- It is *optimistic*, i.e. it thinks the goal is closer than it really is
- Ensures that A* doesn't miss any promising paths
  - If both $g(n)$ and $h(n)$ are low, $f(n)$ will be low and A* will explore state n before considering more expensive paths

Theorem 3.1 (Optimality of A*)**.** *A* is optimal iff* $h(n)$ *is admissible*

Theorem 3.2 (Optimal Efficiency of A*)**.** *Among all* optimal search algorithms *that start from the* same initial state *and use the* same heuristic function*, A* expands the* fewest *states*

A* with Multi Path Pruning

- Optimal if $h(n)$ is *consistent*

Consistent (Monotone) Heuristic

- A heuristic function $h(n)$ is *consistent* iff
  1. $h(n)$ is admissible
  2. For every two neighbouring states n_1 and n_2, $h(n_1) \leq g(n_1, n_2) + h(n_2)$
- If there are more than 1 edge from n_1 to n_2, the inequality must hold for *all* the edges
- A consistent heuristic is admissible

Theorem 3.3 (Optimality of A* with Multi Path Pruning)**.** *If* $h(n)$ *is consistent, A* with multi path pruning is optimal*

3.4 Constructing Heuristics

Example Heuristics for 8-Puzzle

- *Manhattan distance heuristic*: the sum of Manhattan distances (L1) of the tiles to their goal positions
- *Misplaced tile heuristic*: the number of tiles that are *not* in their goal positions

Constructing an Admissible Heuristic

1. Define a *relaxed problem* by simplifying or removing constraints on the original problem
2. Solve the relaxed problem *without search*
3. The cost of the optimal solution to the relaxed problem is an admissible heuristic for the original problem

Constructing Admissible Heuristics for 8-Puzzle

- 8-puzzle rule: a tile can move from square A to B if A and B are adjacent and B is empty
- Manhattan distance heuristic: a tile can move from square A to B if A and B are adjacent ~~and B is empty~~
- Misplaced tile heuristic: a tile can move from square A to B ~~if A and B are adjacent and B is empty~~

Dominating Heuristics

- $h_1(n)$ *dominates* $h_2(n)$ iff
  1. $\forall n, h_1(n) \geq h_2(n)$
  2. $\exists n, h_1(n) > h_2(n)$

Theorem 3.4. *If $h_1(n)$ dominates $h_2(n)$, A^* with $h_1(n)$ never expands more states than A^* with $h_2(n)$ for any problem.*

4 Game Tree Search

4.1 Variations on A*

A* on Games

- There are issues where we need to adjust A*, e.g.
  - The possible positions in a game level create an enormous state space
  - Games characters can't move while A* is being calculated
  - Not all positions in a level have equal value
  - The game level can change over time

Iterative A*

- Uses the heuristic alone to move the NPC while A* is being calculated
  - Adjust the heuristic path to the A* path once the calculation is complete
  - Recalculate A* periodically to adjust to the player's movements

Navigation Meshes

- Performing A* on a huge level is expensive and impractical
- Divide the navigable space into larger sections, and perform A* on this section space
- Once in the target section, perform A* within the section to reach the goal

Subgoals

- To make A* calculations easier, have the AI navigate through subgoal states (e.g. running from one cover position to another)
- Reduces search space to subgoal states

Influence Maps

- Each position on the map is marked with a utility value indicating how worthwhile that position is (needs to be quick to calculate)
- Can create companion AI behaviour to keep NPCs from getting stuck on walls (e.g. spots close to wall have low value)

Challenges with A* for Games

- Changing state space
  - A* considers objects to be static, so moving objects can get in the way
  - Need to find ways to manage multiple moving object
- Common sense
  - A* assumes that path cost and heuristic cost to the goal are the only measures of a location's worth
  - Unwanted behaviour can occur if the AI ignores its surroundings (e.g. going off a cliff)

4.2 Adversarial Game AI

Assumptions

- So far, our search problems have assumed that the agent has complete control of the environment
  - State does not change unless the agent changes it
- Such assumptions may not be reasonable
  - *Stochastic environment* (e.g. weather, traffic accidents)
  - *Other agents* whose interests conflict with ours
  - Search can find a path to a goal state, but the actions might not lead to the goal since the state can be *changed by other agents*

Generalizing Search for External Changes

- Need to handle state changes that are not in the control of the main agent
- **Game tree search**
  - 2+ agents
  - All agents act to maximize their own gain
    - Might not have a positive effect on *our* gain
  - Can generalize this approach to deal with other external factors (e.g. behaviour coming from the environment)

Game Playing State-of-the-Art

- Checkers (International Tiao Qi): solved
- Chess (International Xiang Qi): Deep Blue
- Go (Wei Qi): AlphaGo

The Nature of Games

- There are 2+ agents making changes to the world (i.e. the state)
- Each agent has their own interests and goals
  - The goals may be different
- Each agent *independently* tries to alter the world to maximize its own benefit
  - Cooperation can occur, but only if it benefits both parties

General Games

- How we play depends on how we think the other player will play
- And vice versa
- Joint dependency

Five Properties of Games

1. Two player
   - Algorithms for 2-player can be extended to multi-player
   - Multi-player games can involve alliances

2. Finite
    - Finite number of states and moves from each state
    - For infinite games, we can apply heuristic cutoffs
    - When the game is too large, finite becomes as bad as infinite, and heuristic cutoffs need to be used
3. Zero-sum (or constant sum)
    - Fully competitive: total payoff to all players is constant, i.e. if one player gets a higher payoff, the other player gets a lower payoff
    - E.g. Poker: we win what our opponent lose
    - Nonzero sum games can be cooperative, where some outcomes are preferred by both players
4. Deterministic
    - No chance involved
    - No random elements (e.g. dice, random cards, etc.)
5. Perfect information
    - All aspects of the state are fully observable
    - E.g. no hidden cards

Extensive Form Two-Player Zero-Sum Games

- What we should do depends on what the other player does
- *Strategic/Normal form games*: single move each
    - E.g. rock-paper-scissors
- *Extensive form games*: extend over multiple moves, where players act alternatively
    - E.g. chess, checkers, etc.

4.3 Game Trees and Minimax

Two-Player Zero-Sum Game Definition

- Two *players*: "Max" and "Min"
    - We are Max, and our opponent is Min
- Set of *positions* P (states of the game)
- A *starting position* $p \in P$ (where game begins)
- *Terminal positions* $T \subseteq P$ (where game can end)
- Set of directed edges $E_{\max}$ between some positions (Max's *moves*)
- Set of directed edges $E_{\min}$ between some positions (Min's *moves*)
- *Utility* or *payoff function* $U : T \to \mathbb{R}$ (how good is each terminal state for player Max)

Two-Player Zero-Sum Game Intuition

- Each position also specifies whose turn it is

  - Game ends when some terminal state $\in T$ is reached
- Utility function and terminals replace goals
  - Max wants to maximize the terminal payoff
  - Min wants to minimize the terminal payoff
- Zero-sum since for each terminal node $t \in T$, Max gets $U(t)$ and Min gets $-U(t)$
  - If the sum is a constant C, then Min gets $C - U(t)$

Functions to Support Game Tree Search

- The initial state `START`
- `player(p)` returns the player whose turn it is (in position p)
- `actions(p)` returns the set of legal moves for `player(p)` in position p
  - Defines the directed edges $E_{\max}$ and $E_{\min}$
- `result(p, m)` returns the new position after making move m
  - New position also specifies new player
  - In some games, a player might get multiple moves in succession
- `terminal(p)` returns True if the game is over in position p
- `utility(p)` returns Max's payoff in a terminal position p

Game Tree

- Similar in structure to search trees, but has:
  - Layers that reflect alternating moves between Max and Min
  - Since a full game tree is large (e.g. 10^{40} for chess), during the game tree search, we examine a smaller subtree of the full game tree (i.e. down to a certain depth, called *search horizon*)
    - We want the result to be large enough to determine what move to make
- Max (i.e. our player) does not determine which terminal state is reached
  - After Max moves to a state, Min decides the subsequent state
  - The terminal state is reached through the moves of both Max and Min
- Thus, Max must have a *strategy*
  - Must know (or compute) what to do for each possible move of Min
  - One sequence of moves will not suffice; Max should have a plan for what move to make in response to *each* play by Min

Minimax Strategy

- Assume that the other player will *always play their best move*
  - We want to play a move that will minimize the payoff that could be gained by the other player, which will maximize our payoff
  - Minimize the damage the other player can do
- If Min plays poorly given some circumstances, there might be a better strategy than minimax (i.e. a strategy that gives us a better payoff)

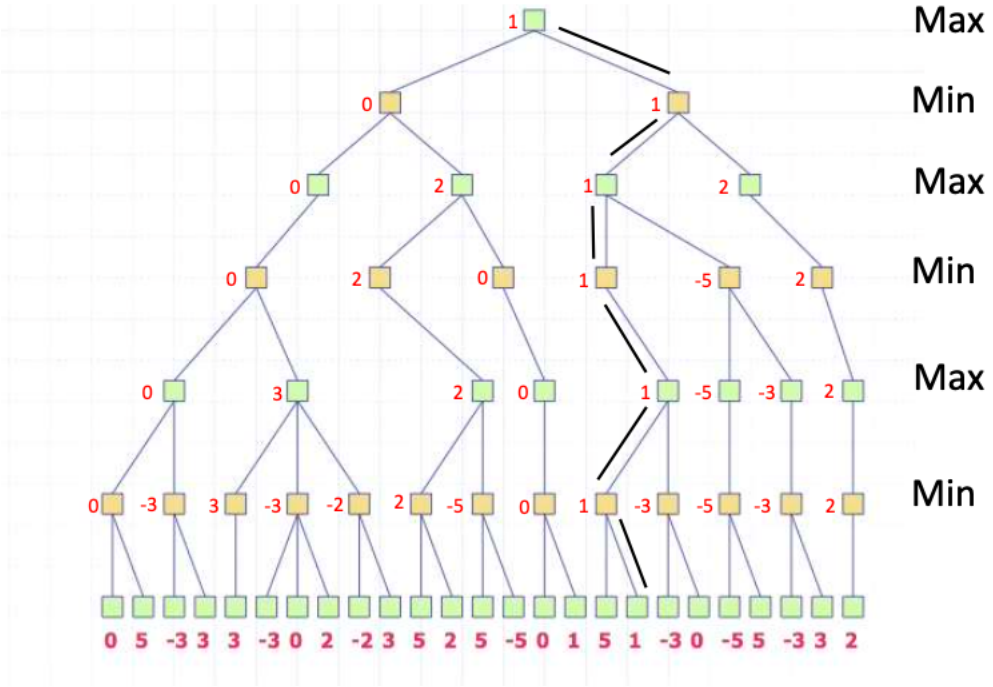
- In the absence of such knowledge, minimax "plays it safe"
- Minimax gives Max the *maximum [minimum payoff over all possible strategies Min could play]*

Minimax Strategy Payoffs

- The terminal nodes have utility values
- Compute a "utility" for non-terminal states by assuming both players always play their *best move*

Minimax Strategy

- Build full game tree (all leaves are terminals)
  - Label terminal nodes with utilities
  - Only feasible for smaller games
- Back values *up* the tree
  - $U(t)$ is defined for all terminals
  - $U(n) = \min\{U(c) : c \text{ is a child of } n\}$ if n is a Min node
  - $U(n) = \max\{U(c) : c \text{ is a child of } n\}$ if n is a Max node
- Max always plays a move to change the state to the highest valued child
- Min always plays a move to change the state to the lowest valued child
- The values $U(n)$ labelling each state n are the values that Max will achieve in that state if both Max and Min play their best moves
- If Min plays poorly, Max could do better, but never worse
- E.g.



4.4 Depth-First Minimax

Depth-First Implementation

- The game tree is exponential in size, which can be too large to store in memory
- We can save space by computing the minimax values with a depth-first implementation of minimax
- Space complexity is linear in the depth of the game tree
- Game tree needs to have finite depth (else the recursive calls will never stop)
- Expands $\mathcal{O}(b^d)$ states, which is both a *best* and *worst* case scenario
  - Must traverse the entire search tree to evaluate all options
  - Different from regular search, where we could just stop upon finding 1 goal state
- Can create a variation that concatenates the (move, value) tuples to return a sequence of moves, not just the first move

Pseudocode

```
1   DFM(pos): # Return best move for player(pos) and MAX's value for pos
2     if terminal(pos):
3       return best_move, utility(pos)
4     if player(pos) == MAX: value = -inf
5     if player(pos) == MIN: value = inf
6     for move in actions(pos):
7       next_pos = result(pos, move)
8       next_move, next_val = DFM(next_pos)
9       if player(pos) == MAX and value < next_val:
10        value, best_move = next_val, move
11      if player(pos) == MIN and value > next_val:
12        value, best_move = next_val, move
13    return best_move, value
```

4.5 Game Tree Pruning

Alpha-Beta Pruning

- It is not necessary to examine the entire tree to make correct Minimax decisions
  - If we've already explored another branch and seen a better outcome, we can rule out a bad move
- While performing depth-first search of a game tree:
  - At any given node N in the tree, the values generated for the children of N may prove that we will *never* reach N in a minimax strategy
  - Depending on the value returned from exploring a child of N, we can stop generating or evaluating any further children of N
  - The other children can be *pruned*
  - Still, using DFS, we need to evaluate *some* children in order to prune others
- Two types of pruning (cuts)
  - Pruning of children of max nodes (α-cuts)
    - α is the highest value found so far (Max wants this)

  - Pruning of children of min nodes (β-cuts)
    * β is the lowest value found so far (Min wants this)

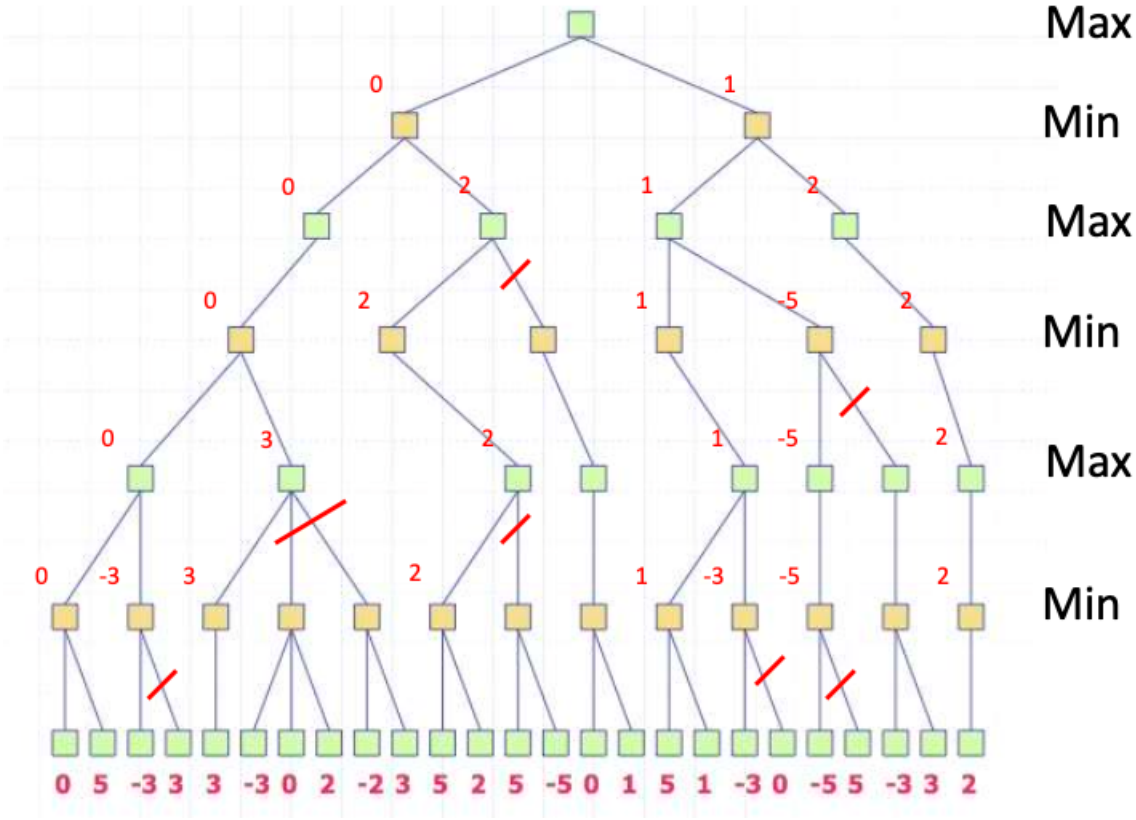
Cutting Max Nodes (Alpha Cuts)

- At a max node n:
  - n's parent chooses its minimum child
  - β is the lowest value found so far by n's parent, from previously-explored siblings of n
  - Value of β is fixed
  - α is the highest value found so far among *children* of n, changes as children of n are examined
- If α becomes $\geq \beta$, then we can stop expading the children of n
  - Min will never choose to move to n since it would choose one of n's lower-valued siblings first

Cutting Min Nodes (Beta Cuts)

- At a min node n:
  - n's parent chooses its maximum child
  - α is the highest value found so far by n's parent, from previously-explored siblings of n
  - Value of α is fixed
  - β is the lowest value found so far among *children* of n, changes as children of n are examined
- If β becomes $\leq \alpha$, then we can stop expading the children of n
  - Max will never choose to move to n since it would choose one of n's higher-valued siblings first

Alpha-Beta Example



Alpha-Beta Pruning Implementation

```
AlphaBeta(pos, alpha, beta): # return best move for player(pos) and MAX's value for pos
   best_move = None
   if terminal(pos):
      return best_move, utility(pos)
   if player(pos) == MAX: value = -inf
   if player(pos) == MIN: value = inf
   for move in actions(pos):
      next_pos = result(pos, move)
      next_move, next_val = AlphaBeta(next_pos, alpha, beta)
      if player(pos) == MAX:
         if value < next_val: value, best_move = next_val, move
         if value >= beta: return best_move, value # beta cut
         alpha = max(alpha, value)
      if player(pos) == MIN:
         if value > next_val: value, best_move = next_val, move
         if value <= alpha: return best_move, value # alpha cut
         beta = min(beta, value)
   return best_move, value
```

- Initial call:

```
      AlphaBeta(START, -inf, inf)
```

Effectiveness of Alpha-Beta Pruning

- With no pruning, we have to explore $\mathcal{O}(b^d)$ nodes
- If the move ordering for the search is optional (i.e. the bset moves are searched first), the number of nodes we need to search using alpha-beta pruning is $\mathcal{O}(b^{d/2})$
- We could theoretically search twice as deep with alpha-beta pruning

4.6 Extensions to Game Tree Search

Ordering Moves

- For MAX nodes, the best pruning occurs if the *best* move for MAX (child yielding lowest value) is explored first
  - Triggers beta cut and returns early
- For MIN nodes, the best pruning occurs if the *best* move for MIN (child yielding lowest value) is explored first
  - Triggers alpha cut and returns early
- We can use heuristics to estimate the value, and then choose the child with highest/lowest heuristic value
  - E.g. for chess, capturing a piece is considered good, and moving away a threatened piece is considered good

Rational Opponents

- May want to compute full strategy ahead of time

- If our opponent has unique optimal choices, this is a single branch through the game tree
- If there are ties, our opponent could choose any one of the "tied" moves, so we must store strategy for each subtree
- Space is an issue for this approach

Depth Limit

- All real games are too large to enumerate tree
- We must limit depth of search tree
  - Can't expand all the way to terminal nodes
  - We must make *heuristic estimates* about the values of the non-terminal positions where we terminate the search
  - These heuristics are called *evaluation functions*
  - Evaluation functions are a combination of learned and hard-coded rules

Example Heuristics in Games

- Tic-tac-toe: [# of length-3 runs that are left open for player A] - [# of length-3 runs that are left open for player B]
- Alan Turing's function for chess: $h(n) = A(n)/B(n)$ where $A(n)$ is the sum of the point value for player A's pieces and $B(n)$ is the sum of the point value for player B's pieces
- Evaluation functions can be specified as a weighted sum of features: $h(n) = w_1 f_1(n) + w_2 f_2(n) + \cdots + w_k f_k(n)$
  - The weights can be learnt

Large Search Problems

- Similar ideas can be used in heuristic search
- Often, we can't expect A* to reach a goal with one search
- We search to some limited path (or limited number of nodes) and then select the first move on the best path we have seen so far
  - Called *online* or *realtime* search
- Guarantees of optimality are lost, but we reduce computational/memory expense

Expectimax Search

- Minimax might play it too safe, which might lead to worse outcomes
- We can consider probabilistic opponents, where our opponent chooses its moves by chance
- Useful when our opponent is "nature", or when there are chance moves in the game (e.g. throwing a dice)
- Now MAX wants to pick a node that maximizes the *expected value*
- *Expectimax search*: compute the average score
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but which move will be picked is uncertain

  - At chance nodes we calculate *expected value*

- Can extend Expectimax to deal with a combination of MAX/MIN/CHANCE nodes

Expectimax Search Pseudocode

```
Expectimax(pos): # Return best move for player(pos) and MAX's value for pos
best_move = None
if terminal(pos):
   return best_move, utility(pos)
if player(pos) == MAX: value = -inf
if player(pos) == CHANCE: value = 0
for move in actions(pos):
   next_pos = result(pos, move)
   next_val, next_move = Expectimax(next_pos)
   if player == MAX and value < next_val:
      value, best_move = next_val, move
   if player = CHANCE:
      value = value + prob(move) * next_val
   return best_move, value
```

5 Constraint Satisfaction Problems (CSP)

5.1 Intro

The search algorithms so far had no knowledge of the state representation (black box)

- For each problem, we have to design a new state representation (i.e. a new search algorithm)
- We can have a general state representation that works well for many different problems
- We can have specialized search algorithms that operate efficiently on a general state representation
- The class of problems that can be represented with this specialized representation are **CSPs** (Constraint Satisfaction Problems)
- CSPs are a special class of search problems with a *uniform and simple* state representation

CSPs

- State representation: *factored* representation of state
- Algorithms: *general purpose* for particular types of constraints
  - Not problem-specific heuristics
- Main idea: eliminate large parts of the search space all at once by identifying variable/value combinations that violate the constraints

State Representation

- Idea: represent states as a vectors of feature values
- We have:
  - A set of k **features** (or **variables**)
  - Each variable has a **domain** of different values
  - A **state** is specified by an assignment of a value for each variable
    - E.g. size = {small, medium, large}, fur = {short, long}, etc.
  - A **partial state** is specified by an assignment of a value to *some* of the variables
- Problem is to search for a set of values for the features so that the values satisfy some conditions (constraints), i.e. a *goal state* specified as conditions on the vector of feature values

Example: Sudoku

- 81 variables, each representing the value of a cell
- Domain of values:
  - For cells that are already filled in, the domain is the singleton set containing the predefined value
  - The domain of other variables is the set $\{1, 2, \cdots, 9\}$
  - State: any completed board given by specifying the value in each cell (1-9)
    - Not necessarily legal
    - No blank spots
  - Partial state: some incomplete filling out of the board
  - Solution: a value for each cell satisfying the constraints:
    1. No cell in the same column can have the same value

2. No cell in the same row can have the same value
3. No cell in the sub subsquare can have the same value

Finding Solution

- We do *not* care about the sequence of moves needed to get to the goal state
- We only care about finding a feature vector that satsifies the goal

Formalization of a CSP

- A CSP consists of
  - A set of **variables** $V_1, \ldots, V_n$
  - For each variable a (finite) **domain** of possible values $\text{Dom}[V_i]$
  - A set of **constraints** $C_1, \ldots, C_m$
  - A **solution** to a CSP is an **assignment** of a value to all of the variables such that *every constraint is satisfied*
  - A CSP is unsatisfiable if no solution exists
- Each variable can be assigned any value from its domain, i.e. $V_i = d$ where $d \in \text{Dom}[V_i]$
- Each constraint C:
  - Has a set of variables it is defined over, called its **scope**
    - E.g. $C(V_1, V_2, V_4)$ is a constraint over the variables V_1, V_2, V_4. The scope of C is $\{V_1, V_2, V_4\}$
  - Acts as a boolean function that maps variable assignments to true/false
    - True: assignment satisfies constraint; false: assignment violates constraint
- Types of constraints
  - *Unary* constraints: over one variable (e.g. $C(X) : X = 1$)
  - *Binary* constraints: over two variables (e.g. $C(X, Y) : X + Y < 6$)
  - *Higher-order* (n-ary) constraints: over 3 or more variables

Example: Sudoku

- Variables: $V_{11}, V_{12}, \ldots, V_{99}$
- Domains:
  - $\text{Dom}[V_{ij}] = \{1, 2, \ldots, 9\}$ for empty cells
  - $\text{Dom}[V_{ij}] = \{k\}$ for filled cells with value k
- Constraints:
  - Row constraints: All-Diff$(V_{i1}, V_{i2}, \ldots, V_{i9})$ for $i = 1, 2, \ldots, 9$
  - Column constraints: All-Diff$(V_{1j}, V_{2j}, \ldots, V_{9j})$ for $j = 1, 2, \ldots, 9$
  - Subsquare constraints: All-Diff$(V_{ij}, \ldots, V_{i+2,j+2})$ for $i, j \in \{1, 4, 7\}$

5.2 Backtracking

Intuition

1. Build a solution by searching through the space of partial assignments
2. If we falsify a constraint during the process of building up a solution, we can immediately reject the current partial assignment
    - All extensions of this partial assignment will falsify that constraint

CSP as a Search Problem

- Initial state: empty assignment
- Successor function: a value is assigned to any unassigned variable, which does not cause any constraint to return false
- Goal test: the assignment is complete

Backtracking Algorithm

- Generic backtracking algorithm
    1. Pick a variable
    2. Pick a value for it
    3. Test the constraints that we can
    4. If a constraint is unsatisfied we backtrack
    5. Otherwise we set another variable
    6. We're done when all the variables are set
- Pseudocode

```
1         BT(Level):
2            if all variables assigned then # Found a solution
3               print value of each variable
4               return (for more solutions) or exit (for one solution)
5            V := PickUnassignedVariable()
6            Assigned[V] := TRUE
7            # Check all possible values for V
8            for d := each member of Domain(V) do
9               Value[V] := d
10              ConstraintsOK = TRUE
11              # Check all constraints over V for current choice d
12              for each constraint C such that V is a variable of C and all other variables of
                     C are assigned do
13                 if C is not satisfied by the set of current assignments then
14                    ConstraintsOK = FALSE
15              if ConstraintsOK == TRUE then
16                 BT(Level + 1) # Recursive call
17           Assigned[V] := FALSE # Undo as we have tried all of V's values
18           return
```

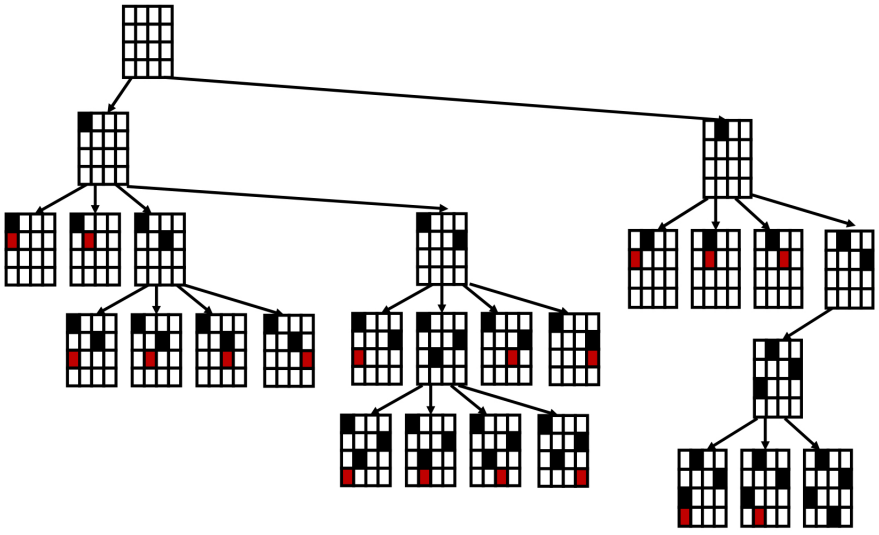
Backtracking Search

- The algorithm searches a tree of partial assignments
    - The root has the empty set of assignments

  - Children of a node are all possible values of some (any) unassigned variable
  - Search stops descending if the assignments on path to the node violate a constraint
- Heuristics are used to determine
  - The order in which variables are assigned, i.e. `PickUnassignedVariable()`
  - The order of values tried for each variable
- The choice of the next variable can vary from branch to branch
  - E.g. under assignment $V_1 = a$, we might choose to assign V_4 next; under assignment $V_1 = b$, we might choose to assign V_2 next
  - Dynamically chosen variable ordering

Example: N-Queens

- Problem: place N queens on an $N \times N$ chess board so that no queen can attack any other queen
- Naive problem formulation
  - N variables (N queens)
  - N^2 values for each variable representing the positions on the chessboard
  - Value i is the ith cell counting from the top-left as 1, going left to right, top to bottom
  - This representation has $(N^2)^N$ states
- Better modelling
  - N variables Q_i, one per row
  - Value of Q_i is the column the queen in row i is placed, with possible values $\{1, \ldots, N\}$
  - This representation has N^N states
  - Constraints:
    - No two queens in the same column: $Q_i \neq Q_j$ for $i \neq j$
    - Diagonal constraints: $|Q_i - Q_j| \neq |i - j|$ for $i \neq j$
- Backtracking search space (better model)



Problems with Plain Backtracking

- A variable could have no possible value, but we won't detect this until we try to assign it a value
- Leads to the idea of *constraint propagation*

5.3 Constraint Propagation (Inference)

Constraint Propagation

- Refers to the technique of "looking ahead" at the yet-unassigned variables in the search
- Try to detect obvious failures (i.e. that we could detect efficiently)
- Even if we don't detect an obvious failure, we might be able to eliminate some possible part of the future search
- Propagation can be applied before the search begins
- Propagation is an *inference step* that needs some resources (i.e. time)
  - If propagation is slow, this can slow the search down to the point where using propagation makes finding a solution take longer
  - There is a *tradeoff* between searching fewer nodes, and having a high nodes/second processing rate
- Two main types of propagation
  1. Forward Checking
  2. Generalized Arc Consistency

5.4 Forward Checking

Forward Checking

- An extension of the backtracking search that employs a "modest" amount of propagation (i.e. look ahead)
- When a variable is instantiated, we check all constraints that have *only one uninstantiated variable* remaining
- For that uninstantiated variable, we check all of its values, pruning those values that violate the constraint

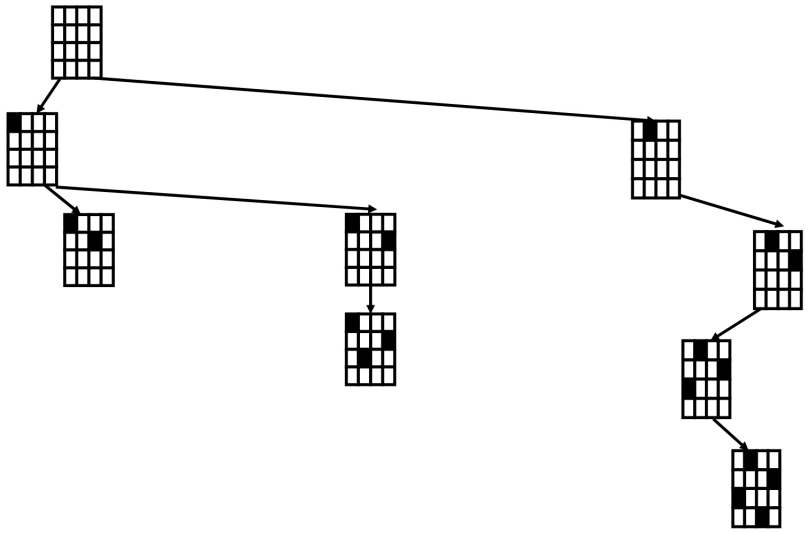
Algorithm

```
1     FCCheck(C, x):
2        # C is a constraint with all its variables already assigned, except for variable x
3        for d := each member of CurDom[x] do
4           if making x = d together with previous assignments to the variables in scope C
                falsifies C then
5              remove d from CurDom[x]
6        if CurDom[x] is empty then return DWO (Domain Wipe Out)
7        else return ok
8
9     FC(Level): # Forward Checking Algorithm
10       if all variables are assigned then
11          print value of each variable
12          return (for more solutions) or exit (for only 1 solution)
13       V := PickAnUnassignedVariable()
14       Assigned[V] := TRUE
15       for d := each member of CurDom(V) do
16          Value[V] := d
17          DWOoccurred := FALSE
```

```
18            for each constraint C over V such that C has only 1 unassigned variable X in its scope
                  do
19               if FCCheck(C, X) == DWO then # domain becomes empty
20                  DWOoccurred := TRUE
21                  break # Stop checking constraints
22            if not DWOoccurred then # All constraints were ok
23               FC(Level + 1)
24            # d is hopeless, restore the domain before backtracking
25            RestoreAllValuesPrunedByFCCheck()
26         Assigned[V] := FALSE # Undo since we have tried all of V's values
27         return
```

- After we backtrack from the current assignment, we must restore the values that were pruned as a result of that assignment
- Must remember which values were pruned by which assignment

N-Queens Example: Search Space



Minimum Remaining Values Heuristics (MRV)

- Always branch on a variable with the *smallest remaining values* (i.e. smallest CurDom)
- If a variable has only 1 value left, that value is forced, so we should propagate its consequences immediately
- Produces skinny trees at the top, meaning that more variables can be instantiated with fewer nodes searched, thus more contraint propagation/DWO failures occur when the tree starts to branch out
- Finds inconsistency much faster

Empirical Running Time

- FC is 100x faster than BT
- FC + MRV is 10000x faster than BT

5.5 Generalized Arc Consistency (GAC)

Arc Consistency (2-Consistency/AC)

- Another for of propagation is to make each arc *consistent*
- $C(X, Y)$ is consistent iff for *every* value of X, there is *some* value of Y that satisfies C
  - E.g. $C(X, Y) : X > Y$, $\text{Dom}(X) = \{1, 5, 11\}$, $\text{Dom}(Y) = \{3, 8, 15\}$
  - For $X = 1$ there is no value of Y that satisfies C, so remove 1 from $\text{Dom}(X)$
  - For $Y = 15$ there is no value of X that satisfies C, so remove 15 from $\text{Dom}(Y)$
  - We obtain $\text{Dom}(X) = \{5, 11\}$, $\text{Dom}(Y) = \{3, 8\}$
- Say we find a value d of variable V_i that is not consistent w.r.t. a constraint, then d is *arc inconsistent*, and we can remove d from the domain of V_i as this value cannot lead to a solution
- Removing a value from a domain may trigger further inconsistency, so we have to repeat this procedure until everything is consistent
  - We put constraints on a queue and add new constraints to the queue as we need to check for arc consistency
- Stronger than forward checking

Generalized Arc Consistency (GAC)

- $C(V_1, \ldots, V_n)$ is GAC w.r.t. variable V_i iff for *every* value of V_i, there exists values of $V_1, \ldots, V_{i-1}, V_{i+1}, \ldots, V_n$ that satisfy C
- Since values are removed from domains during search, a constraint that is GAC w.r.t. a variable may become non-GAC w.r.t that variable
- $C(V_1, \ldots, V_n)$ is GAC iff it is GAC w.r.t. all variables in its scope
- A CSP is GAC iff all of its constraints are GAC
  - We can cycle through all constraints until we stop pruning
- GAC could be performed before we start to search

GAC Algorithm

- Make all constraints GAC at every node of the search space
- Accomplished by removing all arc inconsistent values from the domains of the variables
- Pseudocode

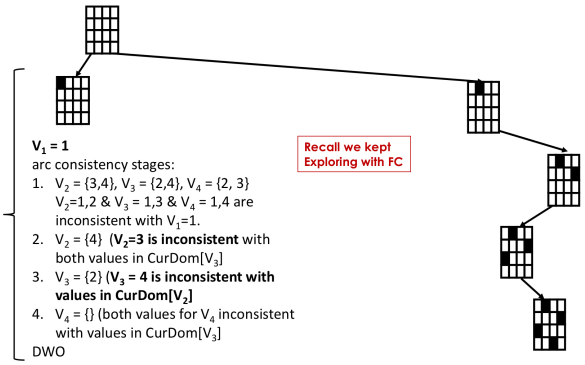
```
1   GAC(Level): # Maintain GAC Algorithm
2     if all variables are assigned then
3       print value of each variable
4       return (for more solutions) or exit (for only 1 solution)
5     V := PickAnUnassignedVariable()
6     Assigned[V] := TRUE
7     for d := each member of CurDom(V) do
8       Value[V] := d
9       prune all values of V != d from CurDom[V] # don't consider other values
10      for each constraint C whose scope contains V do
11        put C on GACQueue
12      if GAC_Enforce() != DWO then
```

```
13                GAC(Level + 1)
14             RestoreAllValuesPrunedFromCurdoms()
15          Assigned[V] := FALSE
16          return
17
18       GAC_Enforce():
19          # GAC-Queue contains all constraints such that one of whose variables
20          # has had its domain reduced. At the root of the search tree,
21          # we first run GAC_Enforce with all constraints on GAC-Queue
22          while GACQueue not empty do
23             C = GACQueue.extract()
24             for V := each member of scope(C) do # Check each variable
25                for d := CurDom[V] do # Check each value
26                   find an assignment A for all other variables in scope(C) such that C(A
                         union {V = d}) is true
27                   if A not found then # d is GAC inconsistent
28                      CurDom[V] = CurDom[V] - d
29                      if CurDom[V] is empty then
30                         empty GACQueue
31                         return DWO
32                      else  # enqueue affected constraints
33                         push all constraints C* such that V is in scope(C*) and C* is not
                               already in GACQueue onto GACQueue
34          return TRUE
```

Enforce GAC

- A **support** for $V = d$ in constraint C is an assignment A to all of the other variables in scope(C) such that $A \cup \{V = d\}$ satisfies C
  - A is what the algorithm's inner loop looks for
  - We could keep track of supports
- Rather than searching for a satisfying assignment to C containing $V = d$, we check to see if the current support is still valid
- Finding a support for $V = d$ in constraint C is $\mathcal{O}(2^k)$ time in the worst case, where k is the *arity* of C, i.e. $|\text{scope}(C)|$
- For some constraints, this computation can be done in polynomial time
  - E.g. For the All-Diff constraint, we can check if $V_i = d$ has a support in the current domains of the other variables in polynomial time using graph algorithms
  - No need to examine all combinations of values for the other variables when looking for a support

N-Queens Example: Search Space



6 Uncertainty

6.1 Probability

Uncertainty

- An agent may not observe everything in the world
- An action may not have its intended consequences
- Therefore an agent needs to
  - Reason about their uncertainty
  - Decide based on their uncertainty

Probability

- Frequentists
  - Probability is *objective*
  - Compute probabilities by counting frequency of events
  - E.g. probability of heads for coin is probability of heads in history
- Bayesians
  - Probability is *subjective*
  - Start with *prior* beliefs and *update* beliefs based on new evidence
  - E.g. probability of heads for coin is probability of heads in the agent's previous experience

Random Variable

- A random variable
  - Has a **domain** of possible values
  - Has an associated **probability distribution**, which is a function from the domain to $[0, 1]$
- E.g.
  - Random variable: the alarm is sounding
  - Domain: {true, false}
  - Pr(The alarm is sounding = true) = 0.1
  - Pr(The alarm is sounding = false) = 0.9

Binary Random Variables

- Capital letter denotes an unassigned rv
- Lowercase letter denotes a variable and its value
- E.g. let X be a Boolean rv
  - $\Pr(x)$ denotes $\Pr(X = \text{True})$
  - $\Pr(\neg x)$ denotes $\Pr(X = \text{False})$
  - $\Pr(X)$ denotes the distribution $\Pr(x), \Pr(\neg x)$

Axioms of Probability (Let A and B be Boolean rvs)

1. Every probability is between 0 and 1
$$0 \leq \Pr(A) \leq 1$$
2. Necessarily true propositions have probability 1; necessarily false propositions have probability 0
$$\Pr(\text{True}) = 1 \quad \text{and} \quad \Pr(\text{False}) = 0$$
3. Inclusion-Exclusion principle
$$\Pr(A \vee B) = \Pr(A) + \Pr(B) - \Pr(A \wedge B)$$

Prior and Posterior Probabilities

- **Prior** or **unconditional** probability
$$\Pr(X)$$
  - Likelihood of X in the absence of any information
  - Based on background info or prior experience
- **Posterior** or **conditional** probability
$$\Pr(X \mid Y)$$
  - Likelihood of X given Y
  - Based on Y as evidence

6.2 Probability Rules

Inference Using the Joint Distribution

- Given a joint distribution, we can compute
  - The probability over a subset of the variables
  - A conditional probability

Sum Rule
$$\Pr(A \wedge B) = \Pr(A \wedge B \wedge c) + \Pr(A \wedge B \wedge \neg c)$$

- We sum out every variable that we do not care about

Product Rule
$$\Pr(A \wedge B) = \Pr(A \mid B) \Pr(B)$$
$$\Pr(A \mid B) = \frac{\Pr(A \wedge B)}{\Pr(B)}$$

- Convert between joint and conditional probabilities

Chain Rule
$$\begin{aligned}\Pr(X_n \wedge X_{n-1} \wedge \cdots \wedge X_2 \wedge X_1) &= \prod_{i=1}^{n} \Pr(X_i \mid X_{i-1} \wedge \cdots \wedge X_1) \\ &= \Pr(X_n \mid X_{n-1} \wedge \cdots \wedge X_2 \wedge X_1) \times \cdots \times \Pr(X_2 \mid X_1) \times \Pr(X_1)\end{aligned}$$

- Calculate a joint probability given the marginal and conditional probabilities
- For three variables:
$$\Pr(X_3 \wedge X_2 \wedge X_1) = \Pr(X_3 \mid X_2 \wedge X_1) P(X_2 \mid X_1) P(X_1)$$

Bayes' Rule
$$\Pr(X \mid Y) = \frac{\Pr(Y \mid X) \Pr(X)}{\Pr(Y)}$$

- Flips a conditional probability

6.3 Unconditional and Conditional Independence

Unconditional Independence

- X and Y are **unconditionally independent** iff

$$\Pr(X \mid Y) = \Pr(X)$$
$$\Pr(Y \mid X) = \Pr(Y)$$
$$\Pr(X \wedge Y) = \Pr(X)\Pr(Y)$$

- Observing the value of X *does not* change our belief in the value of Y

Conditional Independence

- X and Y are **conditionally independent** given Z iff

$$\Pr(X \mid Y \wedge Z) = \Pr(X \mid Z)$$
$$\Pr(Y \mid X \wedge Z) = \Pr(Y \mid Z)$$
$$\Pr(X \wedge Y \mid Z) = \Pr(X \mid Z)\Pr(Y \mid Z)$$

- If we already know the value of Z, then observing the value of X *does not* change our belief in the value of Y

Independence vs. Conditional Independence

- Independence $\not\Rightarrow$ Conditional Independence
- Conditional Independence $\not\Rightarrow$ Independence

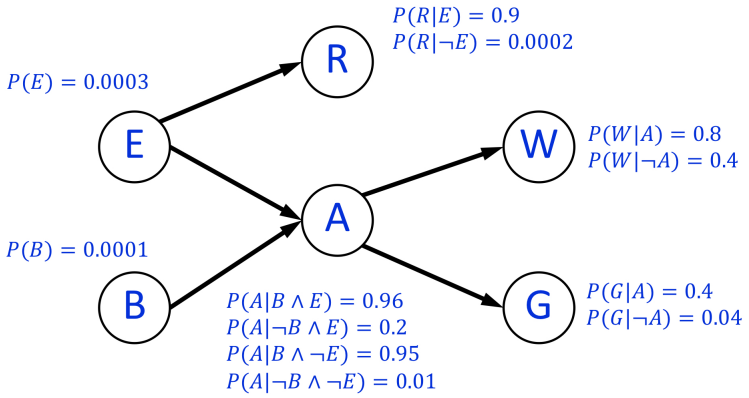
Computational Impact of Independence

- Want to specify the joint distribution of X, Y, Z
  - If X, Y, Z are *not* independent, then we need 7 probabilities (i.e. 2^3 for $\{x, y, z\} \in \{0, 1\}$, excluding the last one which is 1 minus sum of others)
  - If X, Y, Z are independent, then we need 3 probabilities (i.e. $\Pr(X), \Pr(Y), \Pr(Z)$)

6.4 Bayesian Networks

Bayesian Networks

- A *compact* version of the joint distribution
- Takes advantage of the unconditional and conditional independence among the variables



Components of Bayesian Networks

- A direct acyclic graph
- Each node represents a rv
- $X \to Y$ means that X is a parent of Y
- Each node X_i has a conditional probability distribution $\Pr(X_i \mid \text{Parents}(X_i))$ quantifying the effects of X_i's parents on X_i

Semantics of Bayesian Networks

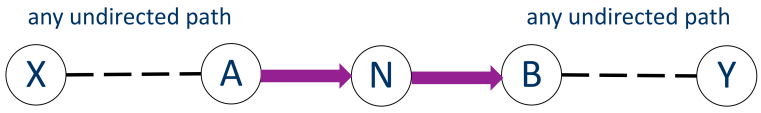
- Represents the joint distribution
- Encoding the conditional/independence relationships

Calculating a Joint Probability

$$\Pr(X_n \wedge \cdots \wedge X_1) = \prod_{i=1}^{n} \Pr(X_i \mid \text{Parents}(X_i))$$

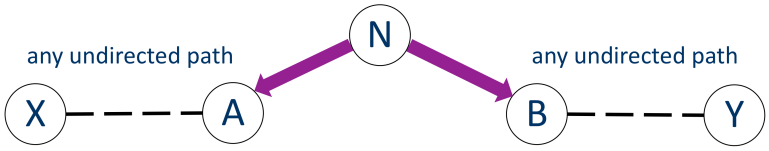
D-Separation

- Want to know whether two variables X and Y independent given the set of observed variables E
- E **d-separates** X and Y iff E *blocks every undirected path* between X and Y
- If E d-separates X and Y, then X and Y are *independent given* E
- Chain situation: observe $\implies$ cut



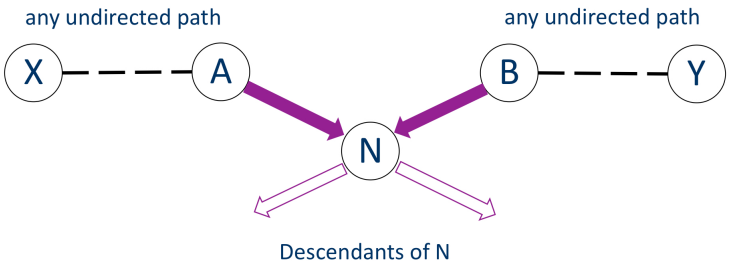
If N is observed,
N blocks the path between X and Y.

- Common parents situation: observe $\implies$ cut



If N is observed,
N blocks the path between X and Y.

- Common children case: *not* observe $\implies$ cut
  - Consider *every* node in the subtree
  - If *at least one* thing is observed in the subtree, then the path is *unblocked*



If N and N's descendants are NOT observed,
they block the path between X and Y.

6.5 Variable Elimination Algorithm

Variable Elimination Algorithm

- Want to compute $\Pr(Q \mid e_1 \wedge \cdots \wedge e_N)$
- *Create a factor* for each node in the Bayesian network
- *Restrict* each evidence variable to its observed value
- Eliminate the hidden variables $h_1, \ldots, h_j, \ldots, h_M$
  - For each hidden variable h_j
  - *Multiply* all the factors that contain h_j to create a new factor f_j
  - *Sum out* h_j from factor f_j
- *Multiply* the remaining factors
- *Normalize* the remaining factor

Create a Factor

- Convert each conditional probability table to a factor
- E.g.

$P(w|a) = 0.8$
$P(\neg w|a) = 0.2$
$P(w|\neg a) = 0.4$
$P(\neg w|\neg a) = 0.6$

w	a	0.8
$\neg w$	a	0.2
w	$\neg a$	0.4
$\neg w$	$\neg a$	0.6

$P(b) = 0.0001$

b	0.0001
$\neg b$	0.9999

Restrict a Factor

- Restrict each evidence variable to its observed value

- E.g.

w	a	0.8
$\neg w$	a	0.2
w	$\neg a$	0.4
$\neg w$	$\neg a$	0.6

Restrict to $A = a$

w	0.8
$\neg w$	0.2

Restrict to $W = \neg w$

0.2

Sum Out a Variable

- Sum out a hidden variable from a factor
- E.g.

w	a	0.8
$\neg w$	a	0.2
w	$\neg a$	0.4
$\neg w$	$\neg a$	0.6

Sum out A

w	1.2
$\neg w$	0.8

Sum out w

2.0

Multiply Factors

- Multiply two factors in an element-wise fashion
- E.g.

w	a	0.8
$\neg w$	a	0.2
w	$\neg a$	0.4
$\neg w$	$\neg a$	0.6

g	a	0.4
$\neg g$	a	0.6
g	$\neg a$	0.04
$\neg g$	$\neg a$	0.96

w	a	g	0.32
$\neg w$	a	g	0.08
w	$\neg a$	g	0.016
$\neg w$	$\neg a$	g	0.024
w	a	$\neg g$	0.48
$\neg w$	a	$\neg g$	0.12
w	$\neg a$	$\neg g$	0.384
$\neg w$	$\neg a$	$\neg g$	0.576

Normalize a Factor

- Convert a factor to a probability distribution

- E.g.

w	1.2
$\neg w$	0.8

Normalized factor

w	0.6
$\neg w$	0.4

6.6 Hidden Markov Model

Inferences in a Changing World

- Since the world evolves over time, we need to reason about a sequence of events
- E.g. weather predictions, stock market predictions, etc.

Example

- Want to know whether it is raining today
- Can only see whether the director comes in with or without an umbrella

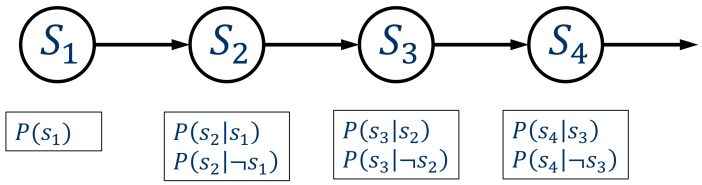
States and Observations

- A series of time steps
- S_t denotes the hidden state at time t
  - E.g. $S_t = 1$ means it rains on day t, and $S_t = 0$ otherwise
- E_t denotes the observation at time t
  - E.g. $E_t = 1$ means the director carries an umbrella on day t, and $E_t = 0$ otherwise

The Markov Assumption

- The future is independent of the past given the present

$$P(S_t|S_{t-1} \wedge S_{t-2} \wedge \cdots \wedge S_1) = P(S_t|S_{t-1})$$

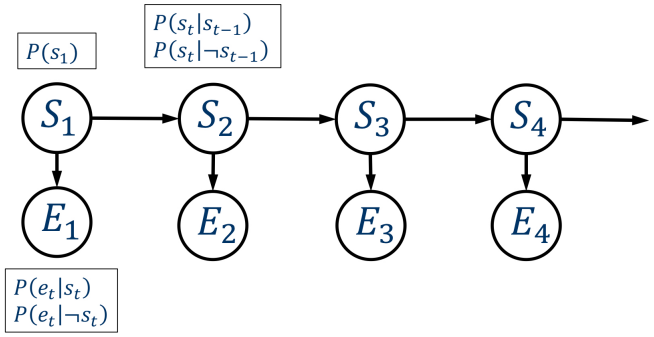


Stationary Process

- The dynamics does not change over time
- $\Pr(S_t \mid S_{t-1})$ is the same for every time step t

Emission Model

- Each state is sufficient to generate its observation



$$P(E_t|S_t \wedge E_{t-1} \ldots \wedge S_1 \wedge E_1) = P(E_t|S_t)$$

- Every E is independent of all previous Es

Hidden Markov Model

- Initial probability $\Pr(s_1)$
- Transition probabilities $\Pr(s_t \mid s_{t-1})$, $\Pr(s_t \mid \neg s_{t-1})$
- Hidden states: S_t
- Observations: E_t
- Emission probabilities: $\Pr(e_t \mid s_t)$, $\Pr(e_t \mid \neg s_t)$

Common Inference Tasks

- Filtering: $\Pr(S_t \mid e_1 \wedge \cdots \wedge e_t)$
  - Given the observations so far, determine what is happening now
  - "Which hidden state am I in right now?"
- Prediction: $\Pr(S_{t+k} \mid e_1 \wedge \cdots \wedge e_t)$
  - Given the observations so far, determine what will happen in the future
  - "Which hidden state will I be in on a future date?"
- Smoothing: $\Pr(S_{t-k} \mid e_1 \wedge \cdots \wedge e_t)$
  - Given the observations so far, update our belief in the past
  - "Which hidden state was I in on a past date?"
  - Since we have more in information in the future, we could get better results
- Most likely explanation: which *sequence of hidden state* is most likely to have generated the observations so far?

Performing Inference Tasks

- An HMM is a Bayesian network
- Can calculate any probability using the variable elimination algorithm
- Use the forward-backward algorithm for filtering and smoothing
- Use the Viterbi algorithm for most likely explanation

Filtering

- Goal is to calculate $\Pr(S_t \mid e_1 \wedge \cdots \wedge e_t)$ for every t
- Recursive algorithm
  - Base case: calculate $\Pr(S_1 \mid e_1)$

$$\Pr(S_1 \mid e_1) = \frac{\Pr(S_1)\Pr(e_1 \mid S_1)}{\Pr(e_1)} = \alpha \Pr(S_1)\Pr(e_1 \mid S_1)$$

    where α is a normalization constant
  - Recursive case: calculate $\Pr(S_k \mid e_1 \wedge \cdots \wedge e_k)$ using $\Pr(S_k \mid e_1 \wedge \cdots \wedge e_{k-1})$

$$\Pr(S_k \mid e_1 \wedge \cdots \wedge e_{k-1}) = \sum_{S_{k-1}} \Pr(S_{k-1} \mid e_1 \wedge \cdots \wedge e_{k-1}) \Pr(S_k \mid S_{k-1})$$

$$\Pr(S_k \mid e_1 \wedge \cdots \wedge e_k) = \alpha \Pr(e_k \mid S_k) \Pr(S_k \mid e_1 \wedge \cdots \wedge e_{k-1})$$

Viterbi Algorithm

- Goal is to find the most likely path through the graph
- The likelihood of a path is the product of:
  - The transition probabilities along the path
  - The probabilities of the observations at each state
- The most likely path to a state on day k is:
  - The most likely path to some state on day $k-1$, followed by
  - A transition to the state on day k

```
Viterbi(E, S, I, T, M): # DP
   # I: initial probabilities, I[i] is the probability of being in state i at time 1
   # T: transition matrix, T[i, j] is the probability of transitioning from state i to state j
   # M: emission matrix, M[i, j] is the probability of emitting observation j from state i
   prob <- matrix(length(E), length(S))
   # prob[i, j] is the probability of being in state j at time i
   prev <- matrix(length(E), length(S))
   # prev[i, j] is the state we came from to get to state j at time i

   # Determine values for time step 0
   for i <- 0, ..., length(S) - 1 do
      prob[0, i] <- I[i] * M[i, E[0]]
      # Initial prob of state i * prob of emitting E[0] from state i
      prev[0, i] <- NIL # No previous setate

   # For time steps 1 to length(E) - 1
   # Find each current state's most likely prior state x
   for t <- 1, ..., length(E) - 1 do
      for i <- 0, ..., length(S) - 1 do
         # Fixing time & current state, find the most likely prior state
         x <- argmax_x in {prob[t - 1, x] * T[x, i] * M[i, E[t]]}
         prob[t, i] <- prob[t - 1, x] * T[x, i] * M[i, E[t]]
         prev[t, i] <- x

   return prob, prev
```

From a CSC373 Perspective:

- Bellman's Equation:

$$OPT(t,s) = \begin{cases} I(s)M(s,E_0), & \text{if } t = 0 \\ \max\limits_{x \in S} \{OPT(t-1,x)T(x,s)M(s,E_t)\}, & \text{if } t > 0 \end{cases}$$

- Final state: $\arg\max\limits_{s \in S} \{OPT(t_f, s)\}$
- Store backpointers to figure out the entire path
- Time complexity: $\mathcal{O}(|E||S|^2)$

7 Knowledge Representation

7.1 Intro

Knowledge Representation

1. Knowledge
    - Information we have about the world we inhabit
    - Mental attitudes and feelings about our environment
2. Representation
    - Symbols standing for things in the world (e.g. words, pictures, mathematical symbols)
    - Not all of our knowledge is symbolically represented (e.g. pixels in a picture)
    - Intelligent agents perform low-level *non-symbolic reasoning* over their perceptual inputs
    - Higher level *symbolically represented* knowledge is also essential

Reasoning

- **Reasoning**: manipulating our symbols to produce new symbols that represents new knowledge
- E.g. deriving a new sentence from existing ones
- Each sentence makes some claim/assertion about our world, which could be true or false
- Reasoning aims to be *truth presenving*
    - If we use reasoning to manipulate a collection of True sentences, we want the newly derived sentence to also be True
    - If our reasoning is truth preserving, we say that it is *sound*
- **Completeness**: our reasoning system is powerful enough to produce *all* sentences that must be true given our current collection of true sentences
    - Requires a formal characterization of "sentence" in order to determine whether we have produced *all* true sentences

Motivation of Knowledge Representation

- Large amounts of knowledge are used to understand the world around us, and to communicate with others
- Reasoning provides an exponential or more compression in the knowledge we need to store
    - E.g. without reasoning we would have to store $1 + 1 = 2$, $1 + 2 = 3$, $1 + 3 = 4$, etc.

Logical Representations

1. They are mathematically precise, thus we can analyse their limitations, their properties, the complexity of inference, etc.
2. They are formal languages, thus computer programs can manipulate sentences in the language
3. They come with both a formal *syntax* and a formal *semantics*
4. Typically, they have well developed *proof theories*, i.e. formal procedures for reasoning to produce new sentences

7.2 First Order Logic

First Order Logic (FOL)

- Two components: **syntax** and **semantics**
- Syntax gives the grammar or rules for forming proper sentences
  - E.g. in programming language, the syntax for an if-statement is

```
if <boolean condition>:
   <expressions>
```

- Semantics gives the meaning of the sentences
  - E.g. semantics of the if-statement: if `<boolean condition>` evaluates to `True`, then we execute `<expressions>`
- Semantics are compositional, i.e. the meaning of complex sentences is determined by the meaning of their parts
  - E.g. in the if-statement, once we know what `<boolean condition>` is testing and what `<expressions>` are doing, we know what the if-statement will do

Basic Semantic Entities of FOL

- Sometimes individual objects are not sufficient, we want to identify tuples of objects that are related to each other. We call these sets of tuples **relations**
- We want to keep track of **functions** over our objects, i.e. $f : D \to D$
  - E.g. for $d \in$ student, we might want a function faculty(d) that gives the faculty the student is registered in
  - More generally we might want $f : D^k \to D$, which has multiple arguments

Basic Syntactic Symbols of FOL

- We can decide what symbols to use
- *Primitive symbols*:

Syntax	**Semantics**
Constant symbols	A particular object $d \in D$
Function symbols	Some function $f : D^k \to D$
Predicate symbols	Some subset of D
Relation symbols	Some subset of D^k

- Additional symbols to connect our basic symbols into sentences:

Syntax	**Semantics**
Equality (commonly used relation)	Subset of $D^2 = \{(d, d) \mid d \in D\}$
Variables (as many as we need) $x, y, z, \ldots$	An object $d \in D$, which particular object can vary
Logical connectives $\wedge, \vee, \neg, \to$	To be introduced
Quantifiers $\forall, \exists$	To be introduced

- E.g. CSC384
  - Objects: students, subjects, assignments, numbers
  - Predicates: difficult(subject), CSMajor(student)

- Relations: handedIn(student, assignment)
- Functions: grade(student, assignment) $\rightarrow$ number

First Order Syntax

- Start with our basic syntactic symbols
  - Note that the function and relation symbols each have specific *arities* (i.e. number of argument)
  - From these we can build up **terms** and **sentences/formulas**
- **Terms** are used as names for objects in the domain (i.e. things in the world)

Terms	**Examples**
Constants	c, Alice, Bob
Variables	x, y, z
Function applications	$f(t_1, \ldots, t_k)$ where t_is are already constructed terms

  - E.g. 5 is a constant term, which is a symbol representing the number 5; $+(5, 5)$ is a function application term, which is a symbol representing the number 10
  - Constants are same as functions that take 0 arguments
  - Constants denote specific objects
  - Functions map tuples of objects to other objects
  - Variables are not yet determined, but they will eventually denote particular objects
- Once we have terms, we can build up *sentences* (*formulas*)
  - *Formulas* represent true/false assertions about terms

Formula	**Examples**
Atomic Formula	$p(t)$ or $r(t_1, \ldots, t_k)$, p is a predicate symbol, r is a k-ary relation symbol, t_i are terms
Negation	$\neg f$, f is a formula
Conjunction	$f_1 \wedge \cdots \wedge f_k$, f_i are formulas
Disjunction	$f_1 \vee \cdots \vee f_k$, f_i are formulas
Implication	$f_1 \rightarrow f_2$, f_1 (called the antecedent) and f_2 (called the consequence) are formulas
Existential	$\exists X.f$, f is a formula and X is a variable
Universal	$\forall X.f$, f is a formula and X is a variable

  - Atoms denote facts that can be *true* or *false* about the world
  - Other formulas generate more complex assertions by composing these atomic formulas, where their truth is dependent on the truth of the atomic formulas in them

7.3 Logic Semantics

Semantics

- A formal mapping from formulas to true/false assertions about our semantic entities
- The mapping mirrors the recursive structure of the syntax, so we can map any formula to a composition of assertions about the semantic entities

Language

- We need to fix the particular first-order language we are going to provide semantics for
- The *primitive* symbols included in the syntax defines the particular language:

$$L(F, P, V)$$

- F is the set of function (and constant symbols)
  - Each symbol $f \in F$ has a particular arity
- P is the set of predicate and relation symbols
  - Each relation symbol $r \in P$ has a particular arity, where the predicate symbols always have arity 1
- V is an infinite set of variables

Primitive Symbols

- An *interpretation* (model) specifies the mapping from the primitive symbols to semantic entities, it is a tuple
$$\langle \boldsymbol{D}, \Phi, \Psi, V \rangle$$
  - $\boldsymbol{D}$ is a nonempty set of objects (domain of discourse)
    - $d \in \boldsymbol{D}$ is an *individual*
    - E.g. $\boldsymbol{D} = \{\underline{\text{Alice}}, \underline{\text{Bob}}, \underline{\text{Craig}}, \underline{\text{Bejing}}, \underline{\text{Toronto}}, \underline{110}, \underline{120}\}$
      - Use underlined symbols to talk about domain individuals
      - Syntactic symbols of the first order language are not underlined
    - Domains are often infinite, but we can use finite models to prime our intuitions
  - Φ specifies the meaning of each primitive function symbol (also handles the primitive constant symbols)
    - Constants (0-ary functions) are mapped to individuals in $\boldsymbol{D}$
      - E.g. $\Phi(\text{client17}) = \underline{\text{Alice}}$
    - 1-ary functions are mapped to particular functions in $\boldsymbol{D} \to \boldsymbol{D}$
      - E.g. $\Phi(\text{rating}) = \text{f\_rating}$, where $\underline{\text{f\_rating(grandhotel)}} = \underline{\text{5stars}}$
    - 2-ary functions are mapped to functions from $\boldsymbol{D}^2 \to \boldsymbol{D}$
      - E.g. $\Phi(\text{distance}) = \text{f\_distance}$, where $\underline{\text{f\_distance(Beijing, Toronto)}} = \underline{110}$
    - n-ary functions are mapped similarly
  - Ψ specifies the meaning of each primitive predicate and relation symbol
    - 0-ary predicates are mapped to true or false
      - E.g. $\Psi(\text{rainy}) = \text{True}$, $\Psi(\text{sunny}) = \text{False}$
    - 1-ary predicates are mapped to subsets of $\boldsymbol{D}$
      - E.g. $\Psi(\text{privatebeach}) = \text{p\_privatebeach}$ (i.e. the subset of hotels that have a private beach), where $\text{p\_privatebeach} = \{\underline{\text{grandhotel}}, \underline{\text{fourseasons}}\}$
    - 2-ary predicates are mapped to subsets of $\boldsymbol{D}^2$ (i.e. sets of pairs of individuals)
      - E.g. $\Psi(\text{available}) = \text{p\_available}$ (i.e. the set of pairs (hotel, week) that have free rooms), where $\text{p\_available} = \{(\underline{\text{grandhotel}}, \underline{\text{week13}}), (\underline{\text{grandhotel}}, \underline{\text{week22}})\}$
    - n-ary predicates are mapped to subsets of $\boldsymbol{D}^n$
  - V specifies the meaning of the variables
    - Takes care of quantification
    - Notation: $V[X/d]$ is a *new* variable assignment function
      - Maps the variable X to the individual d
      - For $Y \neq X$: $V[X/d](Y) = V(Y)$
      - For X: $V[X/d](X) = d$

- The semantics entity that a syntactic symbol maps to is called the **meaning** of the symbol or the *denotation* of the symbol

Symbol	**Semantics**
constant symbol c	$\Phi(c) \in \boldsymbol{D}$ (some particular object)
k-ary function symbol f	$\Phi(f)$ (some particular function $\boldsymbol{D}^k \to \boldsymbol{D}$)
predicate symbol p	$\Psi(p)$ (some particular subset of $\boldsymbol{D}$)
k-ary relation symbol r	$\Psi(r)$ (some particular subset of $\boldsymbol{D}^k$)
variable x	$V(x) \in \boldsymbol{D}$ (some particular object)

Terms

- Given language $L(F, P, V)$ and an *interpretation* $\boldsymbol{I} = \langle \boldsymbol{D}, \Phi, \Psi, V \rangle$ and any term $\boldsymbol{t}$, $\boldsymbol{I}(\boldsymbol{t})$ is the denotation of $\boldsymbol{t}$ under $\boldsymbol{I}$

Term	**Semantics**
constant symbol c	$I(c) = \Phi(c) \in \boldsymbol{D}$ (some particular object)
variable x	$I(x) = V(x) \in \boldsymbol{D}$ (some particular object)
Function application $f(t_1, \ldots, t_k)$	$I(f(t_1, \ldots, t_k)) = \Phi(f)(I(t_1), \ldots, I(t_k))$

- For the function application, first we obtain the denotation of each argument under I, then we apply the function $\Phi(f)$ to these interpreted terms
- Terms always denote individuals under an interpretation I

Formulas

- Formulas will always be True or False under any interpretation I

Formula	**Semantics**
Atomic formula $r(t_1, \ldots, t_k)$	$I(r(t_1, \ldots, t_k)) = \begin{cases} \text{True}, & \text{if } (I(t_1), \ldots, I(t_k)) \in \Psi(r) \\ \text{False}, & \text{elsewise} \end{cases}$

- Ψ maps r to a subset of D^k (i.e. a subset of k-ary tuples of individuals), so the atomic formula is true if its arguments are in the stated relation
- Standard rules for propositional logic

Formula	**Semantics**
$\neg f$	$I(\neg f) = \begin{cases} \text{True}, & \text{if } I(f) = \text{False} \\ \text{False}, & \text{elsewise} \end{cases}$
$f_1 \wedge \cdots \wedge f_k$	$I(f_1 \wedge \cdots \wedge f_k) = \begin{cases} \text{True}, & \text{if } I(f_i) = \text{True for every } i \\ \text{False}, & \text{elsewise} \end{cases}$
$f_1 \vee \cdots \vee f_k$	$I(f_1 \vee \cdots \vee f_k) = \begin{cases} \text{True}, & \text{if } I(f_i) = \text{True for any } i \\ \text{False}, & \text{elsewise} \end{cases}$
$f_1 \to f_2$	$I(f_1 \to f_2) = \begin{cases} \text{True}, & \text{if } I(f_1) = \text{False or } I(f_2) = \text{True} \\ \text{False}, & \text{elsewise} \end{cases}$
$\exists X.f$	$I(f) = \begin{cases} \text{True}, & \text{if for some } d \in \boldsymbol{D}, I'(f) = \text{True where } I' = \langle \boldsymbol{D}, \Phi, \Psi, V[X/d] \rangle \\ \text{False}, & \text{elsewise} \end{cases}$
$\forall X.f$	$I(f) = \begin{cases} \text{True}, & \text{if for all } d \in \boldsymbol{D}, I'(f) = \text{True where } I' = \langle \boldsymbol{D}, \Phi, \Psi, V[X/d] \rangle \\ \text{False}, & \text{elsewise} \end{cases}$

7.4 Models

Models

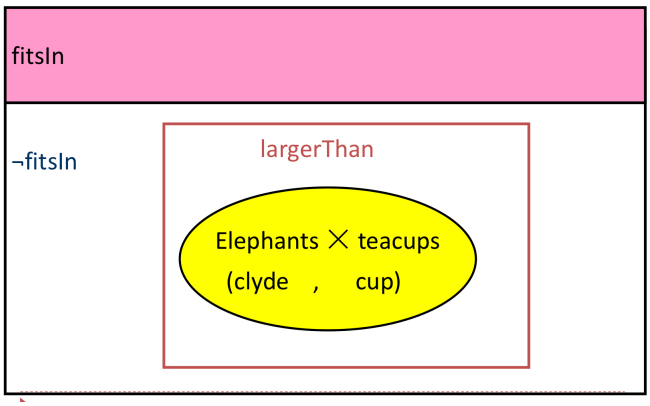
- Let our knowledge base KB consist of a set of formulas
- We say that I is a **model** of KB or that I **satisfies** KB if every formula $f \in KB$ is true under I
- We write $I \models KB$ if I satisfies KB, and $I \models f$ if f is true under I
- When we write KB, we intend that the real world (i.e. our set of theoretic abstraction of it) is one of its models
- Every statement in KB is *true* in the real world
- However not everything true in the real world need to be contained in KB, we might have only incomplete knowledge

Models support reasoning

- Suppose formula f is not mentioned in KB, but is true in every model of KB, i.e. $I \models KB \implies I \models f$, then we say that f is a **logical consequence** of KB or that KB **entails** f
- Since the real world is a model of KB, f must be true in the real world
- Entailment is a way of finding new true facts that were not explicitly mentioned in KB

Example

- elephant(clyde)
  - The individual denoted by the symbol *clyde* is in the set denoted by *elephant*
- teacup(cup)
  - *cup* is a *teacup*
- $\forall X, Y.\text{elephant}(X) \wedge \text{teacup}(Y) \rightarrow \text{largerThan}(X, Y)$
  - For all pairs of individuals, if the first is an elephant and the second is a teacup, then the pair of objects are related to each other by the *largerThan* relation
- $\forall X, Y.\text{largerThan}(X, Y) \rightarrow \neg\text{fitsIn}(X, Y)$
  - for all pairs of individuals, if X is larger than Y, then we cannot have that X fits in Y
- Logical consequences: ¬fitsIn(clyde, cup)



Models and Interpretations

- The more sentences in KB, the fewer models (satisfying interpretations) there are
- The more we write down (as long as it's all true), the "closer" we get to the "real world", since each sentence in KB rules out certain unintended interpretations
- This is called **axiomatizing the domain**

7.5 Proof Procedures

Computing Logical Consequences

- Allows us to reason with our knowledge
  - Represent the knowledge as logical formulas
  - Apply procedures for generating logical consequences
- **Proof procedures**: procedures for computing logical consequences that can be implemented in our programs
  - They do not know or care anything about interpretations
  - They respect the semantics of interpretations

Properties of Proof Procedures

- We write $KB \vdash f$ to indicate that f can be proved from KB
- **Soundness**: $KB \vdash f \rightarrow KB \models f$
  - All conclusions arrived at via the proof procedure are correct, they are logical consequences
- **Completeness**: $KB \models f \rightarrow KB \vdash f$
  - Every logical consequence can be generated by the proof procedure
  - Not necessarily achievable in practice since proof procedures for FOL have high complexity in the worst case

Resolution

- Clausal form
  - A *literal* is an atomic formula or the negation of an atomic formula, e.g. dog(fido), ¬cat(fido)
  - A *clause* is a disjunction of literals, e.g. ¬owns(fido, fred) ∨¬dog(fido) ∨ person(fred)
    - We write (¬owns(fido, fred), ¬dog(fido), person(fred))
  - A *clausal theory* is a conjunction of clauses
- Resolution rule for ground clauses
  - From the two clauses

$$(P, Q_1, \ldots, Q_k)$$
$$(\neg P, R_1, \ldots, R_n)$$

    we infer the new clause

$$(Q_1, \ldots, Q_k, R_1, \ldots, R_n)$$

- To develop a complete resolution proof procedure, for first-order logic, we need
  1. A way of converting KB and f (the query) into clausal form
  2. A way of doing resolution even when we have variables (unification)

Forward Chaining

- If we have a sequence of clauses $C_1, \ldots, C_k$, such that each C_i is either in KB or is the result of a resolution step involving two prior clauses in the sequence, we then have that $KB \vdash C_k$
- Forward chaining is sound so we also have $KB \models C_k$

Refutation Proofs

- We determine if $KB \vdash f$ by showing that a *contradiction* can be generated from $KB \wedge \neg f$
- In this case a contradiction is an *empty* clause ()
- We employ resolution to construct a sequence of clauses $C_1, \ldots, C_m$ such that
  - C_i is in $KB \wedge \neg f$, or is the result of resolving two previous clauses in the sequence
  - $C_m = ()$, the empty clause
- If we can find such a sequence $C_1, C_2, \ldots, C_m = ()$, we have that $KB \vdash f$
- This procedure is sound so $KB \vDash f$
- This procedure is complete so it is capable of finding a proof of any f that is a logical consequence of KB
  - If $KB \vDash f$, then we can generate a refutation from $KB \wedge \neg f$

Example: want to prove likes(clyde, peanuts) from

(elephant(clyde), giraffe(clyde)) (1)
(¬elephant(clyde), likes(clyde, peanuts)) (2)
(¬giraffe(clyde), likes(clyde, leaves)) (3)
¬likes(clyde, leaves) (4)

- Forward chaining proof:

(3) + (4) → ¬giraffe(clyde) (5)
(5) + (1) → elephant(clyde) (6)
(6) + (2) → likes(clyde, peanuts) (7)

- Refutation proof:

¬likes(clyde, peanuts) (5)
(5) + (2) → ¬elephant(clyde) (6)
(6) + (1) → giraffe(clyde) (7)
(7) + (3) → like(clyde, leaves) (8)
(8) + (4) → () (9)

7.6 KB Conversion

Conversion to Clausal Form

1. Eliminate implications
2. Move negations inwards (and simplify ¬¬)
3. Standardize variables
4. Skolemize
5. Convert to Prenex form
6. Distribute conjunctions over disjunctions

7. Flatten nested conjunctions and disjunctions
8. Convert to clauses

Example

$$\forall X.p(X) \rightarrow ((\forall Y.p(Y) \rightarrow p(f(X,Y))) \wedge \neg (\forall Y.\neg q(X,Y) \wedge p(Y)))$$

Eliminate Implications

- Rule: $A \rightarrow B \equiv \neg A \vee B$

$$\forall X.\neg p(X) \vee ((\forall Y.\neg p(Y) \vee p(f(X,Y))) \wedge \neg (\forall Y.\neg q(X,Y) \wedge p(Y)))$$

Move Negations Inwards

- $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- $\neg(A \vee B) \equiv \neg A \wedge \neg B$
- $\neg\forall X.f \equiv \exists X.\neg f$
- $\neg\exists X.f \equiv \forall X.\neg f$
- $\neg\neg A \equiv A$

$$\forall X.\neg p(X) \vee ((\forall Y.\neg p(Y) \vee p(f(X,Y))) \wedge (\exists Y.q(X,Y) \vee \neg p(Y)))$$

Standardize Variables

- Rename variables so that each quantified variable is unique

$$\forall X.\neg p(X) \vee ((\forall Y.\neg p(Y) \vee p(f(X,Y))) \wedge (\exists Z.q(X,Z) \vee \neg p(Z)))$$

Skolemize

- Remove existential quantifiers by introducing *new function symbols*
- Consider the example $\exists Y.\text{elephant}(Y) \wedge \text{friendly}(Y)$, which asserts that there is some individual that is both an elephant and friendly
  - To remove the existential, we *invent* a name for this individual, say a. This is a new constant symbol *not equal to any previous constant symbols* to obtain

    $$\text{elephant}(a) \wedge \text{friendly}(a)$$

  - Need to ensure that the introduced symbol is *new*
- Consider the example $\forall X \exists Y.\text{loves}(X,Y)$, which claims that for every X there is some Y that X loves
  - For each X, the Y could be different
  - Need to use a function that mentions *every universally quantified variable that* **scopes the existential**
  - Can replace the existential Y by a function of X

    $$\forall X.\text{loves}(X, g(X))$$

    where g is a *new* function symbol

    $$\forall X.\neg p(X) \vee ((\forall Y.\neg p(Y) \vee p(f(X,Y))) \wedge (q(X,g(X)) \vee \neg p(g(X))))$$

Convert to Prenex Form

- Bring all quantifiers to the front

- We only have universals at this stage, each with a different name

$$\forall X \forall Y. \neg p(X) \vee ((\neg p(Y) \vee p(f(X,Y))) \wedge (q(X, g(X)) \vee \neg p(g(X))))$$

Distribute Conjunctions over Disjunctions

- Rule: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

$$\forall XY. (\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))) \wedge (\neg p(X) \vee q(X, g(X)) \vee \neg p(g(X)))$$

Flatten Nested Conjunctions and Disjunctions

- Rule: $(A \vee (B \vee C)) \equiv (A \vee B \vee C)$

Convert to Clauses

- Remove quantifiers and break apart conjunctions

$$\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))$$
$$\neg p(X) \vee q(X, g(X)) \vee \neg p(g(X))$$

7.7 Unification

Unification

- *Ground clauses*: clauses with no variables in them
- For ground clauses, we can use syntactic identity to detect when we have a P and $\neg P$ pair
- Want to resolve clauses with variables, e.g.

$$(P(\text{john}), Q(\text{fred}), R(X))$$
$$(\neg P(Y), R(\text{susan}), R(Y))$$

- Once reduced to clausal form, all remaining variables are universally quantified, so each clause represent a set of ground clauses
  - There is a "specialization" of a clause that can be reduced with another
  - E.g. can set Y to john
- We want to be able to match conflicting literals, even when they have variables
- However we don't want to over specialize
- E.g. we have clauses

$$(\neg p(X), s(X), q(\text{fred}))$$
$$(p(Y), r(Y))$$

  - Possible resolvants:
    - $(s(\text{john}), q(\text{fred}), r(\text{john}))$ $\{Y = X, X = \text{john}\}$
    - $(s(\text{sally}), q(\text{fred}), r(\text{sally}))$ $\{Y = X, X = \text{sally}\}$
    - $(s(X), q(\text{fred}), r(X))$ $\{Y = X\}$
  - The last resolvant is the "most general", since the other two are specializations of it
  - We want to keep the most general clause so that we can use it for future resolution steps
- **Unification** is a mechanism for finding a "most general" matching

- First we consider *substitutions*

Substitutions

- A **substitution** is a finite set of equations of the form $V = t$ where V is a variable and t is a term not containing V (i.e. t might contain other variables)
- We can apply a substitution σ to a formula f to obtain a new formula $f\sigma$ by *simultaneously* replacing every variable mentioned in the LHS of the substitution by the RHS, e.g.

$$p(X, g(Y, Z))[X = Y, Y = f(a)] \equiv p(Y, g(f(a), Z))$$

- Substitutions are not applied sequentially
- We can compose two substitutions θ and σ to obtain a new substitution $\theta\sigma$
  - E.g. let

$$\theta = \{X_1 = s_1, X_2 = s_2, \ldots, X_m = s_m\}$$
$$\sigma = \{Y_1 = t_1, Y_2 = t_2, \ldots, Y_k = t_k\}$$

  - To compute $\theta\sigma$, we apply σ to each RHS of θ and then add all of the equations of σ, i.e.

$$S = \{X_1 = s_1\sigma, X_2 = s_2\sigma, \ldots, X_m = s_m\sigma, Y_1 = t_1, Y_2 = t_2, \ldots, Y_k = t_k\}$$

  - Delete any identities, i.e. equations of the form $V = V$
  - Delete any equations $Y_i = s_i$ where Y_i is equal to one of the X_j in θ
  - The final set S is the composition $\theta\sigma$
- Example:

$$\theta = \{X = f(Y), Y = Z\}, \qquad \sigma = \{X = a, Y = b, Z = Y\}$$

  - Compute $\theta\sigma$:

$$\begin{aligned} S &= \theta \text{ with } \sigma \text{ applied to RHS followed by } \sigma \\ &= \{X = f(Y)\{X = a, Y = b, Z = Y\}, Y = Z\{X = a, Y = b, Z = Y\}, X = a, Y = b, Z = Y\} \\ &= \{X = f(b), Y = Y, Z = Y, X = a, Y = b, Z = Y\} \\ &= \{X = f(b), X = a, Y = b, Z = Y\} \\ &= \{X = f(b), Z = Y\} \qquad \text{Since } X \text{ and } Y \text{ were in } \theta \end{aligned}$$

- The empty substitution $\epsilon = \{\}$ is also a substitution, and it acts as an identity under composition
- Substitutions are associative when applied to formulas, i.e.

$$(f\theta)\sigma = f(\theta\sigma)$$

Unifiers

- A **unifier** of two formulas f and g is a substitution σ that makes f and g *syntactically identical*
- Not all formulas can be unified
- Substitutions only affect variables
- The pair

$$p(f(X), a) \qquad p(Y, f(w))$$

cannot be unified as there is no way of making $a = f(w)$ with a substitution

- We typically use UPPER CASE to denote variables, and lower case for constants

Most General Unifier (MGU)

- A substitution σ of two formulas f and g is a **most general unifier (MGU)** if
  1. σ is a unifier
  2. For every other unifier θ of f and g, there must exist a third substitution λ such that $\theta = \sigma\lambda$
     - Every other unifier is "more specialized" than σ
- Example:

$$p(f(X), Z) \qquad p(Y, a)$$

  - $\sigma = \{Y = f(a), X = a, Z = a\}$ is a unifier

$$p(f(X), Z)\sigma = p(f(a), a)$$
$$p(Y, a)\sigma = p(f(a), a)$$

    but it is not an MGU
  - $\theta = \{Y = f(X), Z = a\}$ is an MGU

$$p(f(X), Z)\theta = p(f(X), a)$$
$$p(Y, a)\theta = p(f(X), a)$$

- The MGU is the "least specialized" way of making clauses with universal variables match
- We can compute MGUs
- Intuitively we line up the two formulas and find the first subexpression where they disagree. The pair of subexpressions where they *first* disagree is called the *disaggrement set*
- The algorithm works by successively fixing disagreement sets until the two formulas become syntactically identical

```
1   Find-MGU(f, g): # f, g are formulas
2     k <- 0
3     sigma_0 <- {}
4     S_0 <- {f, g}
5
6     if S_k contains an identical pair of formulas then
7       return sigma_k as the MGU of f and g
8
9     find the disagreement set D_k = {e_1, e_2} of S_k
10    if e_1 = V a variable, and e_2 = t a term not containing V (or vice versa) then
11      sigma_k+1 <- sigma_k {V = t} # compose the additional substitution
12      S_k+1 <- S_k {V = t} # apply the additional substitution
13      k <- k + 1
14      goto 6
15    else
16      return 'f and g cannot be unified'
```

Example: $S_0 = \{p(f(a), g(X)); \quad p(Y, Y)\}$

- $k = 0, \sigma_0 = \{\}$
- $D_0 = \{f(a), Y\}$

- $\sigma_1 = \{\}\,\{Y = f(a)\} = \{Y = f(a)\}$
- $S_1 = S_0\,\{Y = f(a)\} = \{p(f(a), g(X)); \quad p(f(a), f(a))\}$
- $k = 1$
- $D_1 = \{g(X), f(a)\}$, fail since we do not have a variable and a term not containing te variable
- Therefore not unifiable

Example: $S_0 = \{p(a, X, h(g(Z))); \quad p(Z, h(Y), h(Y))\}$

- $k = 0, \sigma_0 = \{\}$
- $D_0 = \{a, Z\}$
- $\sigma_1 = \{\}\,\{Z = a\} = \{Z = a\}$
- $S_1 = S_0\,\{Z = a\} = \{p(a, X, h(g(a))); \quad p(a, h(Y), h(Y))\}$
- $k = 1$
- $D_1 = \{X, h(Y)\}$
- $\sigma_2 = \{Z = a\}\,\{X = h(Y)\} = \{Z = a, X = h(Y)\}$
- $S_2 = S_1\,\{X = h(Y)\} = \{p(a, h(Y), h(g(a))); \quad p(a, h(Y), h(Y))\}$
- $k = 2$
- $D_2 = \{g(a), Y\}$
- $\sigma_3 = \{Z = a, X = h(Y)\}\,\{Y = g(a)\} = \{Z = a, X = h(Y), Y = g(a)\}$
- $S_3 = S_2\,\{Y = g(a)\} = \{p(a, h(g(a)), h(g(a))); \quad p(a, h(g(a)), h(g(a)))\}$
- $k = 3$
- $D_3 = \{\}$
- Therefore MGU is $\sigma_3 = \{Z = a, X = h(Y), Y = g(a)\}$

Non-Ground Resolution

- Resolution of non-ground clauses, i.e.

$$(L, Q_1, Q_2, \ldots, Q_k)$$
$$(\neg M, R_1, R_2, \ldots, R_n)$$

- If there exists σ a MGU for L and M, we infer the new clause

$$(Q_1\sigma, \ldots, Q_k\sigma, R_1\sigma, \ldots, R_n\sigma)$$

- E.g.

$$(p(X), q(g(X)))$$
$$(r(a), q(Z), \neg p(a))$$

  - $L = p(X); M = p(a)$
  - $\sigma = \{X = a\}$

- $R[1a, 2c]\,\{X = a\}\,(q(g(a)), r(a), q(Z))$

- Notation
  - R means resolution step
  - $1a$ means the first (a-th) literal in the first clause
  - $2c$ means the third (c-th) literal in the second clause
  - $\{X = a\}$ is the substitution applied to make the clashing literals identical

Answer Extraction

- We can also answer fill-in-the-blanks questions
- Use free variables in the query where we want to fill in the blanks, e.g.
  - parent(art, jon): is art one of jon's parents?
  - parent(X, jon): who is one of jon's parents?
- To answer the query parent(X, jon), we construct the clause

$$(\neg\text{parent}(X, \text{jon}), \text{answer}(X))$$

- Perform resolution until we obtain a clause consisting of only answer literals

Factoring

- If two or more literals of a clause C have an MGU θ, then $C\theta$ with all duplicate literals removed is called a **factor** of C
- E.g.

$$\begin{aligned} C &= p(X), p(f(Y)), \neg q(X) \\ \theta &= \{X = f(Y)\} \\ C\theta &= (p(f(Y)), \neg q(f(Y))) \qquad \text{is a factor} \end{aligned}$$

- Proof example

$$\begin{aligned} &(p(X), p(Y)) \\ &(\neg p(V), \neg p(W)) \\ &f[1ab]\,\{X = Y\}\,p(Y) \\ &f[2ab]\,\{V = W\}\,\neg p(W) \\ &R[3, 4]\,\{Y = W\}\,() \end{aligned}$$